# Introduction to OpenMP

Cecina Babich Morrow

2024-03-01

The following portfolio contains materials covered in Dr. Christopher Wood's course Beginning OpenMP.

## Introduction to OpenMP

OpenMP gives an interface for parallel programming that works in multiple languages, including C++. Most C++ compilers support OpenMP, including GCC, Clang, and Microsoft Visual C++.

### Hello OpenMP!

The following is a simple example of an OpenMP program in C++:

```cpp
#include <iostream>

int main()
{
    #pragma omp parallel
    {
        std::cout << "Hello OpenMP!\n";
    }

    return 0;
}
```

The `#pragma omp parallel` directive runs the code within the brackets on every core available. Each parallel thread executes all the code in the parallel section.

To compile the program with OpenMP support, we run `g++ -fopenmp hello_openmp.cpp -o hello_openmp`. If we want to optimize the program (which becomes useful for more complex programs), we can instead run `g++ -fopenmp hello_openmp.cpp -o hello_openmp -O3`, compiling the program with OpenMP support and optimization level 3.

Running `export OMP_NUM_THREADS=4` in the bash shell sets the number of threads to 4, determining the number of cores used (you can set this on a line-by-line basis as well, e.g. `OMP_NUM_THREADS=4 ./hello_openmp`). `OMP_NUM_THREADS` resets whenever a new terminal window is opened.

## Pragmas

Pragmas, such as the one seen in our hello world program, are compiler directives that give instructions to the compiler. They are written as `#pragma omp` followed by a command. They are only followed if the program is compiled with OpenMP support. Some common pragmas include:

- `#pragma omp parallel` runs the code in the following brackets on all available cores
- `#pragma omp sections` divides the code in the following brackets into sections, each of which is run in parallel
- `#pragma omp for` creates a loop where different iterations are run on different cores

Note that variables defined within the `parallel` block are thread-local, meaning that each thread has its own copy of the variable.

SIMD (Single Instruction, Multiple Data) is a parallel processing technique that runs the same instruction on multiple data points.

## Loops

OpenMP can parallelize loops within your code as long as the loop iterations are independent of each other. The `#pragma omp for` directive parallelizes the loop iterations, with each thread running a different iteration.

The following C++ code shows a simple parallized loop and outputs the number of iterations each thread performs:

```cpp
#include <iostream>
#include <omp.h>

int main()
{
    #pragma omp parallel
    {
        int nloops = 0; // Each thread will have its own copy of nloops

        #pragma omp for // This pragma will distribute the loop iterations among the threads
        for (int i=0; i<1000; ++i)
        {
            ++nloops;
        }

        int thread_id = omp_get_thread_num(); // Get the thread number

        std::cout << "Thread " << thread_id << " performed "
                  << nloops << " iterations of the loop.\n";
    }

    return 0;
}
```

## Critical code

After running a loop in parallel, we need some way to combine the results from each thread into a global copy that we can then use. This needs to be done carefully, however, so that we don't have multiple threads trying to access the same memory location at the same time, which is known as a data race. We handle this using critical code.

Critical sections of the code are those that must be run by only one thread at a time. This is done with `#pragma omp critical` followed by the code that must be run by only one thread at a time. We want the code within the critical block to be as simple as possible since it won't be run in parallel.

## Reduction

Reduction is a way to combine the results of a parallel loop into a single result. It is difficult to write efficient code for reduction yourself because it requires a critical section, meaning only one thread is updating the result at a time. OpenMP provides a reduction clause that does this for you.

This reduction directive goes at the end of the parallel directive with the form `reduction( operator : variable list )`. For example, `pragma omp parallel reduction( + : nloops)` tells the compiler that `nloops` is a reduction variable that will be formed by adding the values of results from each thread.

## Map/Reduce

Map/Reduce is a programming model for processing and generating large data sets. It is a two-step process: the map step processes the data in parallel, and the reduce step combines the results.

## Overall performance tips

When using OpenMP, it is important to experiment by benchmarking the code to ensure that you are actually improving the performance of your code. The `hyperfine` program is a good tool for this. To check the speed of a program, you can run `hyperfine ./yourfile`. To compare the benchmark performance of two programs, you can run `hyperfine ./program_1 ./program_2`.

## Example: Monte Carlo Pi Estimation

One way to estimate $\pi$ is by using Monte Carlo methods. We use a unit circle, which has an area of $\pi$, inscribed within a square with side length 2, which has an area of 4. If we sample randomly within the square, the ratio of points within the circle to the total number of points will be $\pi/4$.

We will use OpenMP to parallize this process and come up with an estimate for $\pi$.

### Using a critical session

First, we will parallelize the Monte Carlo estimation in a loop and combine the results using a critical session.

```
#include <random>
#include <cmath>
#include <iostream>
#include <omp.h>

int main() {

    std::random_device rd;
    int pts_inside = 0;
    int pts_outside = 0;
```

```
    #pragma omp parallel
    {
        // Set up random number generator stuff
        std::default_random_engine generator(rd());
        std::uniform_real_distribution random(-1.0, 1.0);

        // Thread number
        int thread_id = omp_get_thread_num();

        // Counters within each thread
        int thread_pts_inside = 0;
        int thread_pts_outside = 0;

        #pragma omp for
        for (int i = 0; i < 100000000; i++)
        {
            // Generate random x and y coordinates
            double x = random(generator);
            double y = random(generator);
            // Calculate distance from center
            double distance = std::sqrt(x*x + y*y);

            if (distance <= 1.0)
            {
                thread_pts_inside++;
            }
            else
            {
                thread_pts_outside++;
            }
        }

        double thread_pi_estimate = 4.0 * thread_pts_inside /
          (thread_pts_inside + thread_pts_outside);
        double thread_n_pts = thread_pts_inside + thread_pts_outside;

        #pragma omp critical
        {
            std::cout << "Thread " << thread_id << " estimated value of pi: " <<
            thread_pi_estimate << " (includes " << thread_n_pts << " points)" <<
              std::endl;
            pts_inside += thread_pts_inside;
            pts_outside += thread_pts_outside;
        }

    }

    double pi_estimate = 4.0 * pts_inside / (pts_inside + pts_outside);
    std::cout << "Estimated value of pi: " << pi_estimate << std::endl;

    return 0;
}
```

We check our results after $1 \times 10^8$ iterations of the loop:

```
g++ -fopenmp ../intro_to_openmp/pi_estimate.cpp -o ../intro_to_openmp/pi_estimate -O3
../intro_to_openmp/pi_estimate
```

```
## Thread 10 estimated value of pi: 3.14085 (includes 5e+06 points)
## Thread 15 estimated value of pi: 3.14235 (includes 5e+06 points)
## Thread 7 estimated value of pi: 3.1424 (includes 5e+06 points)
## Thread 2 estimated value of pi: 3.14164 (includes 5e+06 points)
## Thread 19 estimated value of pi: 3.14066 (includes 5e+06 points)
## Thread 14 estimated value of pi: 3.1425 (includes 5e+06 points)
## Thread 6 estimated value of pi: 3.14175 (includes 5e+06 points)
## Thread 8 estimated value of pi: 3.14019 (includes 5e+06 points)
## Thread 16 estimated value of pi: 3.1403 (includes 5e+06 points)
## Thread 1 estimated value of pi: 3.14144 (includes 5e+06 points)
## Thread 3 estimated value of pi: 3.14082 (includes 5e+06 points)
## Thread 0 estimated value of pi: 3.14116 (includes 5e+06 points)
## Thread 9 estimated value of pi: 3.14323 (includes 5e+06 points)
## Thread 18 estimated value of pi: 3.14109 (includes 5e+06 points)
## Thread 11 estimated value of pi: 3.14191 (includes 5e+06 points)
## Thread 13 estimated value of pi: 3.14139 (includes 5e+06 points)
## Thread 12 estimated value of pi: 3.14137 (includes 5e+06 points)
## Thread 4 estimated value of pi: 3.14173 (includes 5e+06 points)
## Thread 17 estimated value of pi: 3.14289 (includes 5e+06 points)
## Thread 5 estimated value of pi: 3.14163 (includes 5e+06 points)
## Estimated value of pi: 3.14156
```

## Using a reduction

Now, instead of using a critical session, we will use a reduction to combine the results of the threads.

```
#include <random>
#include <cmath>
#include <iostream>
#include <omp.h>

int main() {

    std::random_device rd;
    int pts_inside = 0;
    int pts_outside = 0;

    #pragma omp parallel reduction(+ : pts_inside, pts_outside)
    {
        // Set up random number generator stuff
        std::default_random_engine generator(rd());
        std::uniform_real_distribution random(-1.0, 1.0);

        // Counters within each thread
        int thread_pts_inside = 0;
        int thread_pts_outside = 0;
```

```
        #pragma omp for
        for (int i = 0; i < 100000000; i++)
        {
            // Generate random x and y coordinates
            double x = random(generator);
            double y = random(generator);
            // Calculate distance from center
            double distance = std::sqrt(x*x + y*y);

            if (distance <= 1.0)
            {
                thread_pts_inside++;
            }
            else
            {
                thread_pts_outside++;
            }
        }

        double thread_pi_estimate = 4.0 * thread_pts_inside /
          (thread_pts_inside + thread_pts_outside);
        double thread_n_pts = thread_pts_inside + thread_pts_outside;

        pts_inside += thread_pts_inside;
        pts_outside += thread_pts_outside;

    }

    double pi_estimate = 4.0 * pts_inside / (pts_inside + pts_outside);
    std::cout << "Estimated value of pi: " << pi_estimate << std::endl;

    return 0;
}
```

We can compare the estimate results with the critical session method:

```
g++ -fopenmp ../intro_to_openmp/pi_estimate_reduction.cpp -o \
  ../intro_to_openmp/pi_estimate_reduction -O3
../intro_to_openmp/pi_estimate_reduction
```

```
## Estimated value of pi: 3.14171
```

## Comparing performance

Both methods are giving good estimates of $\pi$. We can compare the performance to see which is faster:

```
hyperfine ../intro_to_openmp/pi_estimate ../intro_to_openmp/pi_estimate_reduction
```

```
## Benchmark 1: ../intro_to_openmp/pi_estimate
##   Time (mean ± ):     178.3 ms ±  18.5 ms    [User: 3020.4 ms, System: 2.7 ms]
##   Range (min … max):   170.0 ms … 234.3 ms    12 runs
```

```
##
##   Warning: The first benchmarking run for this command was significantly slower than the rest (234.3
##
## Benchmark 2: ../intro_to_openmp/pi_estimate_reduction
##   Time (mean ± ):     175.4 ms ±   8.9 ms    [User: 2965.6 ms, System: 2.4 ms]
##   Range (min … max):   170.8 ms … 206.9 ms    17 runs
##
##   Warning: Statistical outliers were detected. Consider re-running this benchmark on a quiet PC witho
##
## Summary
##   '../intro_to_openmp/pi_estimate_reduction' ran
##     1.02 ± 0.12 times faster than '../intro_to_openmp/pi_estimate'
```

The reduction method appears slightly faster, although the results are very close.